lib/main/ordered-list-nat.ath

```

# Properties of ordered lists of natural numbers

load "nat-less.ath"
load "list-of.ath"

#........................................................................
# <=L: is a natural number less than or equal to the first element of
# a list of natural numbers (true if the list is empty).

extend-module List {
open N

declare <=L: [N (List N)] -> Boolean

module <=L {
assert empty := (forall x . x <=L nil)
assert nonempty :=
  (forall x y L . x <=L (y :: L) <==> x <= y)

define left-transitive :=
  (forall L x y . x <= y & y <=L L ==> x <=L L)
define before-all-implies-before-first :=
  (forall L x . (forall y . y in L ==> x <= y) ==> x <=L L)
define append :=
  (forall L M x . x <=L L & x <=L M ==> x <=L (L join M))

datatype-cases left-transitive {
  nil =>
  pick-any x y
    assume (x <= y & y <=L nil)
      (!chain-> [true ==> (x <=L nil) [empty]])
| (z :: M) =>
  pick-any x y
    assume (x <= y & y <=L (z :: M))
      conclude (x <=L (z :: M))
        (!chain->
         [(x <= y & y <=L (z :: M))
          ==> (x <= y & y <= z)          [nonempty]
          ==> (x <= z)                   [Less=.transitive]
          ==> (x <=L (z :: M))           [nonempty]])
}

datatype-cases before-all-implies-before-first {
  nil =>
  pick-any x
    assume (forall y . y in nil ==> x <= y)
      conclude (x <=L nil)
        (!chain-> [true ==> (x <=L nil) [empty]])
| (z:N :: L) =>
  pick-any x
    assume i := (forall y . y in (z :: L) ==> x <= y)
      conclude (x <=L (z :: L))
        (!chain-> [(z = z)
                    ==> (z = z | z in L)  [alternate]
                    ==> (z in (z :: L))   [in.nonempty]
                    ==> (x <= z)          [i]
                    ==> (x <=L (z :: L))  [nonempty]])
}

datatype-cases append {
  nil =>
  pick-any M x
    (!chain [(x <=L nil & x <=L M)
             ==> (x <=L M)               [right-and]
             ==> (x <=L (nil join M))    [join.left-empty]])
| (u :: N) =>
  pick-any M x
```

```
    assume (x <=L (u :: N) & (x <=L M))
      (!chain->
       [(x <=L (u :: N))
        ==> (x <= u)                     [nonempty]
        ==> (x <=L (u :: (N join M)))  [nonempty]
        ==> (x <=L ((u :: N) join M))  [join.left-nonempty]])
}
} # <=L

#.....................................................................
# List.ordered: are the natural numbers in a list in order?

declare ordered: [(List N)] -> Boolean

module ordered {
assert empty := (ordered nil)
assert nonempty :=
  (forall L x . ordered (x :: L) <==> x <=L L & ordered L)

define head :=
  (forall L x . ordered (x :: L) ==> x <=L L)
define tail :=
  (forall L x . ordered (x :: L) ==> ordered L)

conclude head
  pick-any L x
    (!chain
     [(ordered (x :: L))
      ==> (x <=L L & ordered L) [nonempty]
      ==> (x <=L L)             [left-and]])

conclude tail
  pick-any L x
    (!chain
     [(ordered (x :: L))
      ==> (x <=L L & ordered L) [nonempty]
      ==> (ordered L)           [right-and]])

define first-to-rest-relation :=
  (forall L x y .  ordered (x :: L) & y in L ==> x <= y)
define cons :=
  (forall L x . ordered L & (forall y . y in L ==> x <= y)
    ==> ordered (x :: L))
define append :=
  (forall L M . ordered L & ordered M &
    (forall x y . x in L & y in M ==> x <= y)
    ==> ordered (L join M))

by-induction first-to-rest-relation {
  nil =>
  pick-any x:N y:N
    assume i := ((ordered (x :: nil)) & y in nil)
      let {not-in := (!chain->
                      [true ==> (~ y in nil) [in.empty]])}
      (!from-complements (x <= y) (y in nil) not-in)
| (z:N :: M:(List N)) =>
  let {ind-hyp := (forall ?x ?y .
                    ordered (?x :: M) & ?y in M ==> ?x <= ?y)}
  conclude (forall ?x ?y .
             ordered (?x :: (z :: M)) & ?y in (z :: M)
             ==> ?x <= ?y)
    pick-any x:N y:N
      assume ((ordered (x :: (z :: M))) & y in (z :: M))
        let {p0 :=
              (!chain->
               [(ordered (x :: (z :: M)))
                ==> (x <=L (z :: M) & ordered (z :: M))
                                   [nonempty]
                ==> (x <=L (z :: M) & z <=L M & ordered M)
                                   [nonempty]
```

```
                  ==> (x <= z & z <=L M & ordered M)
                                     [<=L.nonempty]]);
             p1 :=
              (!chain-> [p0 ==> (ordered M)   [prop-taut]]);
             p2 :=
              (!chain->
               [p0 ==> (x <= z & z <=L M) [prop-taut]
                   ==> (x <=L M)          [<=L.left-transitive]
                   ==> (x <=L M & ordered M)     [augment]
                   ==> (ordered (x :: M))        [nonempty]]);
             p3 := (!chain->
                    [(y in (z :: M))
                     ==> (y = z | y in M)  [in.nonempty]])}
        (!cases (y = z | y in M)
           assume (y = z)
             (!chain-> [p0 ==> (x <= z)  [left-and]
                           ==> (x <= y)  [(y = z)]])
             (!chain [(y in M)
                      ==> (p2 & y in M)     [augment]
                      ==> (x <= y)          [ind-hyp]]))
}

conclude cons
  pick-any L x
    let {p := (forall ?y . ?y in L ==> x <= ?y)}
    assume (ordered L & p)
      (!chain->
       [p ==> (x <=L L) [<=L.before-all-implies-before-first]
          ==> (x <=L L & ordered L    )    [augment]
          ==> (ordered (x :: L))           [nonempty]])

by-induction append {
  nil:(List N) =>
  conclude (forall ?R .
             ordered nil & ordered ?R &
             (forall ?x ?y . ?x in nil & ?y in ?R ==> ?x <= ?y)
             ==> (ordered (nil join ?R)))
    pick-any R
      assume ((ordered nil) & (ordered R) &
              (forall ?x ?y . ?x in nil & ?y in R ==> ?x <= ?y))
        (!chain->
         [(ordered R)
          ==> (ordered (nil join R)) [join.left-empty]])
| (z :: L:(List N)) =>
  let {ind-hyp :=
           (forall ?R .
             ordered L & ordered ?R &
             (forall ?x ?y . ?x in L & ?y in ?R ==> ?x <= ?y)
             ==> (ordered (L join ?R)))}
  conclude
      (forall ?R .
        ordered (z :: L) & ordered ?R &
        (forall ?x ?y . ?x in (z :: L) & ?y in ?R ==> ?x <= ?y)
        ==> (ordered ((z :: L) join ?R)))
    pick-any R:(List N)
      let {A1 := (ordered (z :: L));
           A2 := (ordered R);
           A3 := (forall ?x ?y .
                     ?x in (z :: L) & ?y in R ==> ?x <= ?y)}
      assume (A1 & A2 & A3)
        let {C1 :=  (!chain->
                      [(ordered (z :: L)) ==> (ordered L)  [tail]]);
             C2 := conclude
                      (forall ?x ?y . ?x in L & ?y in R ==> ?x <= ?y)
                     pick-any x:N y:N
                       assume D := (x in L & y in R)
                         (!chain->
                          [D ==> (x in (z :: L) & y in R)  [in.tail]
                             ==> (x <= y)                  [A3]]);
             C3 := conclude (ordered (L join R))
```

```
                    (!chain->
                     [C1 ==> (C1 & (ordered R) & C2) [augment]
                         ==> (ordered (L join R))    [ind-hyp]]);
            C4 := conclude (z <=L R)
                     (!two-cases
                      assume (R = nil)
                      (!chain-> [true ==> (z <=L nil)  [<=L.empty]
                                      ==> (z <=L R)    [(R = nil)]])
                      assume (R =/= nil)
                        let {D1 :=
                              conclude (z in (z :: L))
                                (!chain->
                                 [(z = z)
                                  ==> (z = z | z in L)  [alternate]
                                  ==> (z in (z :: L))   [in.nonempty]]);
                             D2 := (exists ?u ?M . R = (?u :: ?M));
                             D3 := conclude D2
                                      (!chain->
                                       [true
                                        ==> (R = nil | D2)
                                                [(datatype-axioms "List")]
                                        ==> ((R =/= nil) & (R = nil | D2))
                                                              [augment]
                                        ==> D2                [prop-taut]])}
                        pick-witnesses u M for D3
                          (!chain->
                           [true
                            ==> (u in (u :: M))  [in.head]
                            ==> (u in R)         [(R = (u :: M))]
                            ==> (z in (z :: L) & u in R)  [augment]
                            ==> (z <= u)         [A3]
                            ==> (z <=L (u :: M)) [<=L.nonempty]
                            ==> (z <=L R)        [(R = (u :: M))]]))}
         conclude (ordered ((z :: L) join R))
           (!chain->
            [(ordered (z :: L))
             ==> ((z <=L L) & ordered L)     [nonempty]
             ==> (z <=L L)                   [left-and]
             ==> ((z <=L L) & C4)            [augment]
             ==> (z <=L (L join R))          [<=L.append]
             ==> (z <=L (L join R) & C3)     [augment]
             ==> (ordered (z :: (L join R))) [nonempty]
             ==> (ordered ((z :: L) join R)) [join.left-nonempty]
             ])
    }
} # ordered
} # List
```